CS422: Programming Language Design

Elements of Object-Oriented Programming

Grigore Roșu and Mark Hills
grosu@cs.uiuc.edu, mhills@cs.uiuc.edu

University of Illinois at Urbana-Champaign

October 10, 2006

Object-Oriented Languages

Object-oriented (OO) languages are those based around the concepts of:

- The **class**, an abstract entity which specifies data and functionality related to that data; and
- the **object**, an instance of a particular class which actually contains the specified data and provides something for the specified functionality to work on.

OO languages can be seen as an extension of ADT mechanisms from imperative languages like Ada or Modula-2, but with some additional features.

OO Languages – Core Concepts

OO languages generally support the following 3 concepts:

- **Encapsulation**: The capability to contain and partially hide parts of definitions, such as data values held in an object, with an interface to the data; area of most similarity to ADTs
- **Inheritance**: The capability to specialize one or more existing classes by reusing their functionality in a new class
- **Polymorphism**: The capability for objects of different classes to respond to the same request in different ways

Different languages approach these three points in different ways. This gives us our design space.

Our Language: KOOL

We will look at a language named KOOL, the K-based Object-Oriented Language. KOOL is:

- **Dynamic**: The KOOL language is not typed
- **Pure**: All data values are objects; all computation is performed via method calls
- **Imperative**: A standard imperative model is used (versus having a hybrid functional-oo model)

Encapsulation

Encapsulation provides a method of wrapping data and functionality into a related entity:

- Each class specifies data associated to it;
- each class specifies functionality that knows how to operate over that data;
- each object contains an instance of the data specified in the class;
- objects ask other objects to do things for them by calling methods or sending messages.

Basic Encapsulation

At its most basic, encapsulation just allows data members (aka properties or fields) and function members (methods) to be declared as part of a class.

- Note that this does not give us any data hiding;
- this also doesn't give us any "internal" functionality – any object could call a method on any other object

Encapsulation Options

A range of choices about how to handle encapsulation is available in current OO languages:

- are fields visible outside a class?
- can objects of a class see inside other objects of the same class?
- is it possible to set different levels of visibility on fields and methods?
- do the visibility mechanisms interact with the inheritance mechanism?
- do the visibility mechanisms interact with other naming/grouping mechanisms?

Initialization

Many OO languages allow for newly created objects to be initialized automatically, which furthers the "black box" aspect of encapsulation:

- some OO languages encourage standard practices to set up objects – `initialize` methods in Smalltalk, for example
- other OO languages automatically call *constructor* methods (Java, C++, C#) that perform required setup
- some OO languages have a corresponding *destructor* or *finalizer* to do any cleanup

Encapsulation in KOOL

KOOL's approach to encapsulation is:

- All fields are *private* – not visible outside the object that owns them, not even in other objects of the same class
- All methods are *public* – any method defined on the object's class can be called on that object
- Classes have *constructors*, which are automatically called and which are named identically to the class
- Classes do not have *destructors*; KOOL is assumed to be GCed, but no time has been spent on this yet

Inheritance

In OO languages, **inheritance** allows a class to reuse functionality from (usually) another class, without having to rewrite or copy/paste the code.

- The **parent**, **base**, or **super** class is the class being inherited from.
- The **child**, **derived**, or **sub** class is the class doing the inheriting.
- The child class can reuse the functionality from the parent class, including functionality the parent got from its parent, etc.

Types of Inheritance

- Some languages only allow a class to inherit from one other class – this is **single inheritance**.
- Some languages allow a class to inherit from multiple classes, which is **multiple inheritance**.
- Other forms exist, including **mixins** and **traits** – we won't bother with those here

Single Inheritance

Single-inheritance languages only allow a class to inherit from one other class.

- Examples include Java, C#, and Smalltalk
- Generally includes a designated base class from which all other classes inherit (`Object` in Java)
- Semantically cleaner
- Easier to predict behavior
- Easier to translate/compile
- Cannot model some situations

Multiple Inheritance

Multiple inheritance allows a single class to have multiple base classes.

- Examples include C++ and Eiffel
- Classes do not necessarily inherit from another class (in C++, classes need not have a parent class)
- Can model more situations
- Semantically more confusing
- Harder to correctly compile

Problems with Multiple Inheritance

Multiple inheritance is not without its problems. The biggest have to do with *duplication* of functionality and data (so-called *repeated* inheritance):

- Functionality duplication: how do you figure out which method to call when two base classes have methods with the same name? or which field to access when two fields of the same name exist?
- Data duplication: what happens when we inherit the same data member from two different base classes?

Problem: Functionality Duplication

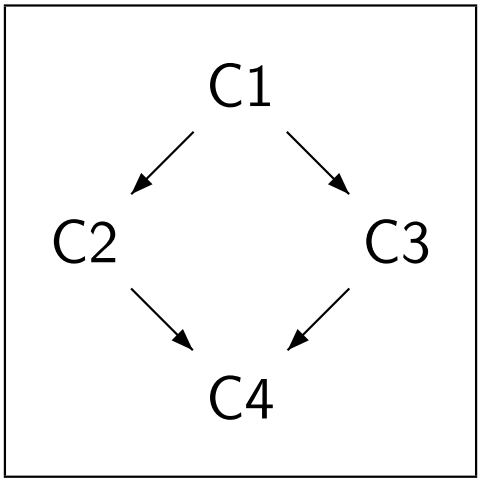
```
1  class CardGame {
2    ...
3    draw() { ... } // draw a card
4    ...
5  }
6
7  class GraphicsObject {
8    ...
9    draw() { ... } // draw the object
10   ...
11 }
12
13 class GraphicalCardGame
14   inherits CardGame, GraphicsObject {
15   ...
16 }
```

Possible Solutions

- Pick one randomly (not a good idea!)
- Require explicit disambiguation (C++)
- Use explicit import and renaming on inheritance (Eiffel)
- Use declaration order (CLOS)
- Use an ordered search of the inheritance graph (Python)

Problem: Data Duplication

- If we have a field defined in C1, how many copies are in C4?
- May want one copy, may want more than one
- Also know as the "diamond problem" (or the more dramatic "diamond of death"!)



Possible Solutions

- Always just include one copy
- Always include all the copies
- Allow programmer to choose (C++)

Note the first two solutions will exclude some valid programs, since we may want just one, or more than one, copy in various situations.

Compromise: Interface Inheritance

Single-inheritance languages like Java and C# allow multiple *interfaces* to be inherited. Interfaces are just named groups of method declarations, generally without bodies and without state (partially finished classes, with some methods left undefined, are usually called *abstract* classes).

- Advantage: specifies that a class supports specific functionality, maybe multiple interfaces; supports polymorphism (below)
- Disadvantage: doesn't allow for reuse, since interfaces don't include method bodies; leads to copy/paste reuse

Overloading and Overriding

Inheritance allows methods in derived classes to **override** methods in base classes:

- New method should have same signature as original method;
- new method hides original method in some cases, based on language rules

A method with the same name but a different signature is said to **overload** that name, giving it multiple meanings:

- Overload rules differ between languages, including what can be a valid overload
- Overload interaction with inheritance also differs between languages – what happens if you overload an inherited name?

Using Inherited Functionality

- Non-overridden methods and fields can be directly referenced (based on visibility rules)
- Base-class methods can usually be called using keywords (`super.method` in Java) or scope resolution (`MyBase::method` in C++)
- These are based on the visibility rules from encapsulation – in Java, `private` methods from a parent aren't suddenly public in a child, for instance, so the child cannot see them.

Alternate Model – Beta

The Beta language uses an alternate model. Instead of calling the method and using `super` to call the parent definition, the definition at the root of the inheritance hierarchy is called first and `inner` is used to call back towards the child.

- Note that this makes calling the child optional!
- Helps to support *behavioral subtyping* – the child cannot ignore the parent, unlike in languages with `super` but not `inner`, so the parent can enforce behavior

Inheritance in KOOL

- The KOOL language uses *single inheritance*, with all classes eventually inheriting `Object`
- KOOL does not have interfaces, which don't really make sense anyway with dynamic languages of this sort
- A `super` keyword provides access to functionality from the parent class
- Field lookup starts in the current class and works backwards towards the `Object` class
- Methods can be *overridden*, but not *overloaded*

Polymorphism

Polymorphism is the third key feature of OO languages. At its core, polymorphism provides the mechanism for letting objects of different classes figure out how to do things themselves – a `Circle` object would draw itself differently than a `Triangle` object, for instance, but I should be able use use them both as `Graphics` objects.

Methods of Enabling Polymorphism

Polymorphism can be enabled in two major ways:

- **Structurally**: especially in dynamically typed or untyped languages, if I have a reference to an object I can send it a message, and if it understands it then it can take action
- **Subtype Polymorphism** – especially in statically typed languages, if class A inherits from class B then an object of class A should be usable wherever an object of class B can be used, but I should be able to "customize" class A to respond differently to the same requests

Virtual Methods

The key to getting subtype polymorphism to work is that the method to invoke should be choosable *at runtime*. This way the behavior can vary, based on the type of the object. Often methods that allow this are called **virtual** methods.

- **Virtual** comes from virtual function table, the structure often used to resolve the correct function to invoke;
- in some languages (like Java) all methods are virtual;
- in other languages (like C++) methods can be static or virtual; the former has better performance;
- this way of selecting methods is also called **dynamic dispatch**.

Method Selection

In the presence of dynamic dispatch, the way to choose the proper method to invoke varies from language to language:

- The proper virtual method is chosen at runtime based on the dynamic type of the object;
- the proper static method is chosen at compile time based on the static type of the variable holding the object (pointer/reference);
- some languages have more involved models – *multimethods* in CLOS or *predicate dispatch* (a current research topic).

Polymorphism in KOOL

- KOOL is dynamic and allows structural polymorphism; if a method is defined with the same name and arity, it will be used
- KOOL uses dynamic dispatch by default, and does not allow static dispatch (except via the `super` keyword)

Self reference

Many OO languages allow reference to the current object

- The `self` keyword is used in many languages, including KOOL
- The `this` keyword is also common, especially in C++ and its progeny

Exceptions

Many OO languages, including KOOL, provide *exceptions*:

- A `try` block contains the code that may throw an exception
- A `catch` clause contains the exception handling code
- A `finally` clause to perform any needed cleanup (in some languages, like Java, but not C++, KOOL, or some others)
- Exceptions are often used by the system, as well, to signal error conditions: in KOOL, a `nil` message target, an invalid method name, etc
- Optional or required handling: optional in KOOL, C++, C#, required in Java

Runtime Type Inspection

KOOL provides the ability to check the type of an expression at runtime, executing different code based on the type. This is also available in many other OO languages, including Java and C++.